Fiche TD0 : Graphes de flot de contrôle

Exercice 1 (*) *Chemin faisable*

On considère le code suivant :

```c
int main()
{
    int i = 1, s = 0;
    while (i <= 1000)
    {
        s = s+i;
        i = i+1;
    }
    printf("%d", s);
}
```

1. Etablir son graphe de flot de contrôle.
2. Déterminer le nombre de chemins dans ce graphe puis le nombre de chemins faisables.
3. Quel entier sera affiché après exécution de ce programme ?

Exercice 2 (*) *GrapheS de flot de contrôle*

On considère le programme suivant :

```c
int branche(int c, int x)
{
    if (c==1) x = x+1;
    if (c==2) x = x-1;
    return x;
}
```

1. En établir le graphe de flot de contrôle.
2. Déterminer le nombre de chemin dans ce graphe. Y a-t-il des chemins infaisables ? Si oui, lesquels ?
3. Pour chacun des chemins faisables, déterminer un jeu de test permettant de le couvrir.
4. Ecrire en C un programme de même spécification que branche (qu'on déterminera) utilisant la structure de contrôle if ... then ... else. Dessiner son graphe de flot de contrôle, déterminer les chemins faisables dans ce graphe et commenter.

Remarque : Comme ici illustré par cet exemple, on peut produire plusieurs programmes dont la spécification est la même sans que la structure de leurs graphes de flot de contrôle coïncident (ce qui est loin d'être une surprise !). Partant de cette constatation, on peut se poser une question : peut on "minimiser" le graphe de flot de contrôle d'un programme de sorte à "l'optimiser" ?

C'est dans l'optique de répondre à ce type de question que les graphes de flot de contrôle ont initialement été conceptualisés par Frances Allen, première femme lauréate du prix Turing, lors de travaux sur la conception et l'optimisation de compilateurs.

Exercice 3 (**) *Limites des couvertures naïves*

On considère la fonction suivante :

```
float inverse_somme(int debut, int fin, int tab[])
{
    int i = debut;
    float somme = 0;
    while (i <= fin)
    {
        somme = somme + tab[i];
        i = i + 1;
    }
    return 1/somme;
}
```

1. Inférer la spécification probablement souhaitée pour cette fonction.
2. Donner son graphe de flot de contrôle.
3. Indiquer quel est le chemin couvert par l'entrée $\{\mathsf{debut} = 0, \mathsf{fin} = 2, \mathsf{tab} = \{1,2,3\}\}$. Ce chemin couvre-t-il tous les sommets ? Couvre-t-il tous les arcs ?
4. Le test précédent suffit-il à tester de manière satisfaisante le comportement de la fonction inverse_somme ?

Exercice 4 (**) *Calcul de conditions de chemin*

La fonction suivante prend en entrée deux entiere x et n, supposés appartenir à $\mathbb{N}$, et renvoie l'entier x^n.

```
int puissance(int x, int n)
{
    int s = 1;
    int p = n;
    while (p >= 1)
    {
        if (p % 2 != 0)
        {
            p = p-1;
            s = s*x;
        }
        s = s*s;
        p = p/2;
    }
    return s;
}
```

1. Construire le graphe de flot de contrôle de cette fonction.
2. Déterminer un ensemble de chemins permettant de couvrir tous les sommets. Couvre-t-il tous les arcs ?
3. Déterminer tous les chemins passant au plus deux fois (deux exclus) dans la boucle tant que.
4. Pour chacun des trois chemins précédents, calculer par exécution symbolique les conditions de chemin associées. En déduire une instance de test dans chacun des cas.
5. Que penser de cette fonction ?

Exercice 5 (***) *Couverture MC/DC*

On considère l'extrait de code suivant :

```
if ((A||B) && C) {instructions;}
else {instructions;}
```

où A, B, C sont des expressions booléennes atomiques (sans connecteur). En présence d'une condition σ faisant intervenir des sous-conditions et d'un ensemble des tests pour cette condition, on dit que :

- La couverture des conditions est vérifiée si chaque sous-condition de σ est évaluée au moins une fois à vrai et une fois à faux lors des tests.
- La couverture des décisions est vérifiée si la condition σ est évaluée au moins une fois à vrai et une fois à faux lors des tests.
- La couverture des conditions/décisions modifiées (Modified Condition/Decision Coverage) est vérifiée si le jeu de tests est tel que : σ est évaluée au moins une fois à vrai et une à faux et chaque sous-condition de σ est évaluée au moins une fois à vrai et une fois à faux tout en ayant une influence sur le résultat final. Autrement dit, pour que ce critère de couverture soit satisfait, d'un cas de test à l'autre, si il n'y a qu'une seule des sous-conditions dont la valeur change alors la valeur de σ doit changer.

1. Dans le cas ci-dessus, donner un nombre minimal de tests satisfaisant la couverture des conditions.
2. Proposer un jeu de test minimal pour la couverture des décisions.
3. Donner un jeu de test minimal permettant d'obtenir la couverture MC/DC pour cette condition.
4. Donner une couverture MC/DC avec aussi peu de tests que possible de la condition $((u = 0) \vee (x > 5)) \wedge ((y < 2) \vee (z = 0))$. Comparer le nombre de tests effectués avec celui que l'on aurait fait si on avait testé exhaustivement toutes les valeurs possibles des sous-conditions.

Remarque : Le test exhaustif de toutes les valeurs possibles pour les sous-conditions d'une condition se heurte à un problème de combinatoire : si il y a n sous-conditions, il faut 2^n tests pour couvrir tous les cas. La couverture MC/DC est un critère de couverture des conditions utilisé dans des normes de test exigeantes (notamment dans l'aérospatiale) permettant de tester intensivement les conditions non atomiques d'un programme en évitant d'avoir un nombre de tests exponentiel en le nombre de sous-conditions : on peut montrer qu'il existe une couverture MC/DC comptant $O(n)$ tests lorsque la condition contient n sous-conditions. Reste évidemment à savoir comment construire ces couvertures !